Idiosyncrasies of System/360 Floating-Point

by Leonard J. Harding, Jr.

Introduction

The following review explores certain aspects of the announced System/360 floating-point feature which have not been adequately described elsewhere. The available knowledge on computational processes and techniques indicate that this facet of System/360 contains many anomalies. Most prominent among these are the results produced by the add, subtract, and compare instructions when the operands are unnormalized, and the effects of not providing a guard digit in the long operand additive operations and multiplication. In the Models 65 and 67 these peculiarities can be eliminated by revisions of their ROS program and minor alterations to the control circuitry.

The University of Michigan Computing Center - August, 1966

The Floating-Point Additive Operations

Some of the most intriguing aspects of the floating-point feature concern the add, subtract and compare instructions. The peculiarities of these operations are inherent in the technique used to form the so-called intermediate sum. The following quotation from the Principles of Operation (Form A22-6821-2) explains how this sum is formed.

> Addition of two floating-point numbers consists of a characteristic comparison and fraction addition. The characteristics of the two operands are compared, and the fraction with the smaller characteristic is right-shifted; its characteristic is increased by one for each hexadecimal digit of shift, until the two characteristics agree. The fractions are then added algebraically to form an intermediate sum ...
>
> The short intermediate sum consists of seven hexadecimal digits and possible carry ... The long intermediate sum consists of 14 hexadecimal digits and possible carry. No guard digit is retained.

It is only what happens after the formation of this intermediate sum which differentiates the twenty add, subtract and compare instructions; in fact, up to this point the same section of the ROS program controls all of these instructions. The following examples illustrate the formation of this intermediate sum. The operand signs indicate the effective signs, so the actual instruction is irrelevant. These examples are based on the Model 65 microprogram and have been checked on a Model 50.

Example 1

Operands	After Preshift
+40 100000 00000000	+40 100000 00000000
-3F FFFFFF FFFFFFFF	-40 $\underline{\text{0FFFFF}}$ $\underline{\text{FFFFFFFF}}$
	+40 000000 00000001

If a guard digit was retained in the long intermediate sum, the result would have been +40 000000 00000000 1, which is 1/16 of the result obtained above. This is an extreme example that would rarely occur in normal computation. It does, however, force the round-off error of floating-point addition to be expressed in a form that is incompatible with the assumptions of most *a priori* error analyses, e.g., Wilkinson [1] and Ortega [2].

Example 2

Operands	After Preshift
+46 000001	+46 000001
+40 123456	+46 $\underline{\text{000000 1}}$
	+46 000001 1

This short operand example illustrates two points. First, if accuracy is to be maintained, it is mandatory that all operands be normalized. For example, when an integer is converted to floating-point format, it must be normalized before it can safely be used in an additive operation. The above situation might occur when computing a logarithm, since normally the logarithms of the exponent and mantissa are computed separately and then added together. Second, if the short operand additions produced long operand results, then in this particular case, the exact result would have been obtained. On most large computers, floating-point addition of single precision operands produces a double precision result, e.g., the IBM 7090. The System/360 breaks this tradition.

<u>Example 3</u>

```
        Operands                    After Preshift
    +40 876543 21012348        +41 087654 32101234
    -41 087654 32101234        -41 087654 32101234
                               -------------------
                               +41 000000 00000000
```

If the instruction was an add or subtract, a significance exception would take place unless masked. For the compare instructions, the condition code would be set to reflect equality. If a guard digit was used, a significance exception would not occur since the result would be

+41 000000 00000000 8; likewise, the compare instructions would not give the erroneous equality indication. These operands are easy to obtain. The first is the result of dividing +41 10ECA8 64202469 by 2, while the second occurs if the same number is halved using the HDR instruction. Note that the last digit of the first operand mantissa is ignored, and hence could be any one of the 16 possible hexadecimal digits. This illustrates the following general result: *a floating-point operand with p leading zeros in the mantissa compares equal to* 16^p *other floating-point numbers*.

Example 4

Operands	After Preshift
+4E 000000 00000000	+4E 000000 00000000
+40 123456 789ABCDE	+4E 000000 00000000
	+4E 000000 00000000

This particular situation is not new. The IBM 7090-94 systems will produce the same erroneous results given the appropriate operands. This example illustrates the following general result: *a floating-point operand with a zero mantissa and a characteristic of N will compare equal to any operand with a characteristic less than or equal to N - 14. Addition* or subtraction of two such operands causes a significance exception. This happens because the mantissa belonging to the larger characteristic is not inspected; zero mantissas are not recognized. With the IBM 7090-94 systems this problem is of little consequence since it is impossible to produce these semi-zeros[1] using normalized floating-point operations. In System/360, however, every time a significance exception is taken one of these semi-zeros is left in the result register[2]. If the computation is resumed without setting this register to a true zero, the entire computation may be lost.

Further, consider the timing aspects of this problem.

```
+4E 123456 789ABCDE          +4E 123456 789ABCDE
+40 000000 00000000          +4E 000000 00000000
                             -------------------
                             +4E 123456 789ABCDE
```

There is nothing wrong with the answer, but in the Model 65 2.8 μsec will be used to preshift the zero mantissa. A true zero is likewise not recognized as such. If a true zero is added to +10 123456 12345678, 3.2 μsec. will be used to preshift the zero mantissa since the characteristic difference

[1]A floating-point operand with a zero mantissa and non-zero characteristic.

[2]If the interruption was masked-off the result register would be set to a true zero.

is 16. The operand with the larger characteristic is taken as the intermediate sum only when the characteristic difference exceeds 16.

Example 5

Operands	After Preshift
+4E 000000 00000001	+4E 000000 00000001
+41 123456 12345678	+4E 000000 00000001
	+4E 000000 00000002

This example illustrates the same points as example 2, but let's consider the timing. Since the characteristic difference is 13 the preshift will require 2.6 µsec. If the result is to be normalized, an additional 2.8 µsec. will be required to finish the operation. If the first operand had been normalized prior to the addition, no preshift or post-normalization would have been necessary. Further, the result would be +41 223456 12345678, not +41 200000 00000000.

There are two further peculiarities of the floating-point feature which affect the additive operations. The most intriguing of these is the significance exception. As has been illustrated, normalized operands are mandatory if accuracy is to be maintained; hence, let's assume normalized operands. Under these conditions, exactly those operations that we know a priori involve no round-off error will cause a significance exception. Further, if the exception is taken a semi-zero will be placed in the result register. An intimately related problem is caused by the fact that no instruction was provided to set a floating-point register to zero. Providing that core storage is not busy, a zero may be loaded in 1.4 μsec. It has been indicated by some that it is easier and faster to subtract the appropriate register from itself. This technique does shorten the instruction stream, but it creates a significance exception. If the exception is masked off the SDR instruction will execute in 1.4 μsec and the result register will be a true zero. Otherwise, a time-consuming interruption will occur and the result register will contain a semi-zero. Everything considered, it would appear that normalized floating-point arithmetic is more safely and economically performed with the significance exception masked off.

The idiosyncrasies of the floating-point additive operations illustrated by these examples may or may not manifest themselves sufficiently often to be bothersome. Their presence, however, would seem to indicate that care will have to be exercised when programming computational processes. The microprogram of the Model 65 could be re-written to avoid all of these peculiarities. The salient points of the revision for the normalized instructions and compare should be:

(1) prenormalization of the operand with the largest characteristic up to the characteristic difference and simultaneous checking for a zero mantissa,

(2) checking of the operand to be preshifted for zero mantissa,

(3) a guard digit for unlike sign additions,

(4) short operand additions produce long operand results.

Assuming this method of operation, all of the above examples would produce the exact result and the extraneous shifting performed in examples 2, 4 and 5 would be eliminated.

The HALVE Instruction

The following programming note from the *Principles of Operation* delineates the defects of this instruction.

> The halve operation differs from the divide operation with the number two as a divisor in the absence of prenormalization and postnormalization and in the absence of a zero-fraction test.

Example 3 of the first section shows that the DD instruction and the HDR instruction will produce different results due to the lack of postnormalization. Note, however, that the compare instruction indicated equality in that example. The Newton-Raphson iteration specialized to the computation of the square root of A is

$$x_{n+1} = 1/2 (x_n + A/x_n).$$

The HDR instruction should not be used here unless you are willing to accept the dangers inherent in unnormalized floating-point numbers.

It should also be taken into account that the hexadecimal base implies that halving and division by two are both subject to round-off error. This is not true of most computers, since halving can be accomplished by characteristic manipulation. Assuming normalized operands, the result of an HDR instruction will be unnormalized if and only if the leading digit of the mantissa is a 1. If the HDR instruction normalized the result, this is the only case in which no round-off error would occur.

In the Model 65, minor revision of the ROS program would make the HDR instruction equivalent to division by two. For a normalized operand the execution time would be increased by at most .4 μsec.

Long Operand Multiplication

The following addition to the most recent version of the *Principles of Operation* (Form A22-6821-2) describes the problem with long operand multiplication.

> Because of the truncation of intermediate results to 14 hexadecimal fraction digits and the introduction of a low-order zero in a subsequent left shift, if any, the low order digit in the result of a long-precision multiplication of normalized operands is not necessarily significant. The truncation error affects only the low-order fraction digit, and the effect of the truncation is predictable.

That this programming note is limited to "normalized operands" is interesting since all floating-point multiplies prenormalize the operands. In general, the low-order product digit will be replaced by zero whenever the product of the two high-order digits of the normalized mantissas can be expressed as one hexadecimal digit, e.g., 1*F=F, 2*7=E, 3*5=F, 4*3=C, etc. Assuming a random distribution of high order digits, the low-order digit of the result is truncated about 20% of the time. In particular, multiplication by 1 is equivalent to truncation to thirteen hexadecimal digits, e.g.,

```
1.  * +40 FFFFFF FFFFFFFF = +40 FFFFFF FFFFFFF0.
```

The loss of the low-order digit also affects multiplication by .5 since .5 = +40 800000 00000000 and 8×1 = 8. Specifically, multiplication by .5 is not equivalent to division by 2 or halving by the HDR instruction. Consider the operand used in example 3, then

```
                          /2 (DD)  =  +40 876543 21012348
+41 10ECA8 64202469       ÷2 (HDR) =  +41 087654 32101234
                          *.5 (MD) =  +40 876543 21012340
```

The results produced by these three operations will be the same when the first digit of the normalized mantissa is <u>not</u> 1. If the first digit is a 1, <u>both</u> multiplication and halving exhibit their peculiarities.

The failure to provide a multiplicative guard digit unnecessarily increases the bound on round-off error by a power of 16. In the Model 67 the guard digit can be obtained by a minor revision of the ROS program. This change increases short and long multiply times by .2 μsec. Short operand times would be increased because the ROS changes must be made at a point in the microprogram which is common to all four multiply instructions.

Bibliography

1. Wilkinson, J. H. Rounding Errors in Algebraic Processes, Prentice-Hall, Englewood Cliffs, N.J., 1963.

2. Ortega, J. M. An Error Analysis of Householder's Method for the Symmetric Eigenvalue Problem, Applied Math. and Stat. Laboratories, Stanford University, Tech. Rep. No. 18, 1962.